Cross-Correlating Spectral Lines

Isaac Domagalski

Astronomy 120

Cross-Correlation

In this lab, one method that you have at your disposal to find the velocity shift in spectral lines from the sun is to compute the cross-correlation of two spectral lines in different images. The method that is used for this is a general method that can be applied to a wide variety of problems, so I'll provide a couple of examples of using the cross-correlation using `numpy`. I will provide the code for the examples so that you can see concrete examples of implementing the cross-correlation.

The cross-correlation is defined as

$$(f \star g)(\tau) \equiv \int_{-\infty}^{\infty} f^*(t)\ g(t+\tau)\ dt \tag{1}$$

where the discrete form of the cross-correlation is defined as[1]

$$(f \star g)[n] \equiv \sum_{m=-N/2}^{N/2-1} f^*[m]\ g[m+n] \tag{2}$$

where the $*$ is the complex conjugation.

In this lab, you are measuring the Doppler velocity by measuring a relative shift between two spectra. Mathematically, if a spectral line is $I[p]$, where p is pixel number[2], then the shifted line is $I[p-P]$, where P is the amount of shift, and the cross-correlation can be written down as

$$(I_p \star I_{p-P})[q] = \sum_{-N/2}^{N/2-1} I[p]\, I[p-P+q] \tag{3}$$

I've dropped the conjugation from the equation since the spectral lines are real quantities. The way to find the shift P from Equation (3) is fairly straightforward to justify. Suppose that we set $q = P$. This is just gives back the sum of the square of the spectral line.

$$I \star I = \sum_{-N/2}^{N/2} I^2[p] \tag{4}$$

[1]Here, I am assuming that N, the total number of samples, is even. If N is odd, then the lower and upper bounds on the summation respectively are $-\lfloor N/2 \rfloor$ and $\lfloor N/2 \rfloor$.

[2]Pixel number should be defined so that the zero pixel is either in the center of the CCD or the spectral line being correlated.

If the q is not equal to the pixel shift, then the value of Equation (3) is going to integrate to something less than than when the when q is equal to P and the spectra are aligned. A visualization of this can be seen in Figure 1. When the two peaks are out of phase, the maximums of the peaks are multiplied by the low tails, and the total area under the product is a lot less than when the two curves are in phase.

Computing the Correlation

There are two ways to compute the cross correlation using the tools available in Python. Both methods that I will present require even spacing between samples, which is why I have been using pixels as my x-axis in the previous explanation instead of wavelength. There are other ways to choose an x-axis for more advanced purposes, but for the scope of this lab, an evenly-spaced pixel grid is the best choice.

The first way to compute the cross-correlation is to note that the cross-correlation can be written in terms of the convolution as

$$f \star g = f^* (-t) * g \tag{5}$$

and to use the `np.convolve` function to cross-correlate. The syntax for this is

```
# The red-shifted spectral line will have the variable name specline_rs
cross_corr = np.convolve(specline[::-1], specline_rs, 'same')

# Create the pixel axis for computing the lag
n_pixels = len(cross_corr)
if n_pixels % 2: # Number of pixels is odd
    shift_axis = np.arange(-n_pixels/2+1, n_pixels/2+1)
else: # Number of pixels is even
    shift_axis = np.arange(-n_pixels/2, n_pixels/2)
```

It is important to note that the first spectral line has been reversed in the above code snippet, and that the actual velocity shift would be computed using the wavelength calibration. The pixel axis is set so that the midpoint `shift_axis[len(shift_axis)/2 + len(shift_axis)%2]` returns zero. This method is a brute-force method of computing the correlation. There is a second method of computing the cross-correlation using Fourier Transforms, which is more advanced and can be found in the appendix at the end of this tutorial.

I haven't actually specified how to measure the lag, and I'm going to mostly leave that as an exercise for you to do on your own. I will say a few things though. The pixel lag due to red shift from solar rotation is less than the actual size of a pixel. Simply finding the maximum is insufficient to find the lag from the cross-correlation. Also, before running the cross-correlation, there are a couple of quick processing steps that you must do before computing the cross-correlation, which improve the accuracy of your measured lag. I'm not going to discuss these steps in their full detail, since that would be giving you the answer for free, but a couple of quick hints to get started are to think about removing background from data, as well as thinking about the fact that the spectral line data is dominated by the window function of the CCD.

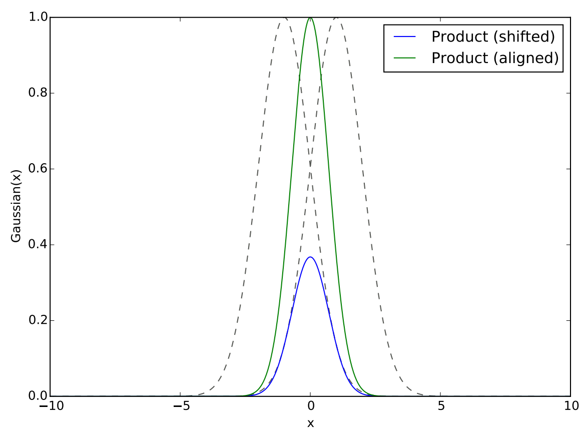

Figure 1: This example shows that when two functions are phase shifted from one another, the integral of their product is less than when the curves are aligned.

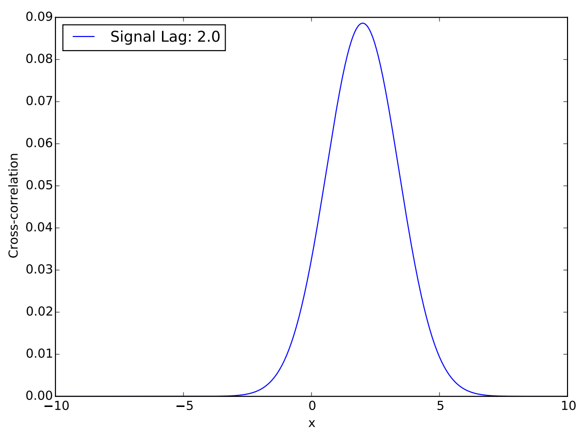

Figure 2: This example measures the separation of the two Gaussian functions of the previous example by cross-correlating them and finding the maximum.

Example: Two Shifted Gaussians

The ability of the cross-correlation to find the lag between two signals can be seen by looking at some examples. The first example that I will present is the cross-correlation of two Gaussian functions that have some distance between the peaks. This example is useful since Gaussian functions are somewhat similar to spectral lines that one would measure in a star (actually, in a star, the spectral line shape is closest to a Voigt profile). The separation in these two Gaussians is much larger than the line shift that you will be measuring from solar data so that it is easy to visualize measuring the lag between them.

Figure 1 shows two Gaussian functions with means of ± 1. The green line shows their product when they are aligned, whereas the blue line shows the product when there is some lag between the two signals. When doing the cross-correlation, you are basically sliding the two functions that you are correlating past each other while integrating them. It is clear that in Figure 1 that if the two peaks are farther apart from each other, then there is less overlap area, and the area under the product goes to zero, but if the functions are nearby each other, then the area under the product is maximized.

While Figure 1 is mainly for showing what happens to products of functions when they are misaligned versus when they are aligned, one can use those peaks to show an example of the cross-correlation in action and measure the distance between the two peaks. The cross-correlation function of the two shifted peaks is plotted in Figure 2. The peak of the correlation is at 2, which is what we would expect since that was the separation between the two functions that we fed it. It should be noted that since the lag between the two Gaussians was very large compared to the pixel size, measuring it from the cross-correlation was done just be finding the max of the cross-correlation. This part of the example diverges from what you will see with your solar spectrum, which has the sub-pixel shift.

Appendix A: Cross-Correlating with the Fourier Transform

While the method described above is easy to compute and will work well on the data that you have collected, it suffers from the fact that the computational complexity (the order-of-magnitude number of computations) is $O\left(N^2\right)$, which is not good for large data sets. There is a way to compute the correlation with $O\left(N \log N\right)$ complexity using Fourier transforms. If you don't know what a Fourier transform is, please see the next appendix at the end of this tutorial for a quick overview.

The cross-correlation has a nice Fourier transform theorem, which is that

$$\mathcal{F}\{f \star g\} = \left(\mathcal{F}\{f\}\right)^* \cdot \mathcal{F}\{g\} \tag{6}$$

where $\mathcal{F}$ is the Fourier transform. This relation is basically the convolution theorem, and given Equation (5), this can easily be seen as an application of the convolution theorem. The fact that the Fourier transform of a cross-correlation is the product of the Fourier transforms of the two input functions is something that can be applied for a much faster computation of the correlation than using `np.convolve`. The Python method of computing the Fast Fourier Transform (FFT) is to use functions from the `numpy.fft` module. Since the FFT algorithm has $O\left(N \log N\right)$ complexity, computing the correlation using the Fourier transform is much more efficient than directly computing it. Since this method is faster, I will be using it in my examples instead of the brute-force method.

Here is a code-snippet illustrating the usage of the Fourier transform to compute the cross-correlation:

```
from numpy import fft # Must import the numpy.fft submodule

# The red-shifted spectral line will have the variable name specline_rs
cross_corr = np.conjugate(fft.fft(specline)) * fft.fft(specline_rs)

# fftshift is necessary, since the inverse transform returns the samples in
the array corresponding to [0, P/2] before the [-P/2, 0] samples.
cross_corr = fft.fftshift(np.real(fft.ifft(cross_corr)))

# Create the pixel axis for computing the lag
n_pixels = len(cross_corr)
if n_pixels % 2: # Number of pixels is odd
    shift_axis = np.arange(-n_pixels/2+1, n_pixels/2+1)
else: # Number of pixels is even
    shift_axis = np.arange(-n_pixels/2, n_pixels/2)
```

The method for computing the lag once you have done the cross-correlation is the same as above. It should be noted that since the input arrays are real, the cross-correlation should return a real array. That being said, the FFT functions in `numpy` return complex values, and for a real array of floats, the imaginary parts are so tiny that they can just be discarded by calling `np.real` after the inverse FFT step in the correlation.

Appendix B: Summary of the Fourier Transform

The Fourier Transform, and it's inverse, of a function $x\left(t\right)$ are defined as

$$X\left(f\right) = \mathcal{F}\{x\left(t\right)\} = \int_{\infty}^{\infty} x\left(t\right) e^{-2\pi i f t} dt \tag{7}$$

$$x\left(t\right) = \mathcal{F}^{-1}\{X\left(f\right)\} = \int_{\infty}^{\infty} X\left(f\right) e^{2\pi i f t} df \tag{8}$$

where if t represents a time variable, then f is the frequency variable. If the signal $x\left(t\right)$ is real, then a useful property of the transform is that $X\left(-f\right) = X^{*}\left(f\right)$. If the function $z\left(t\right) = x\left(t\right) * y\left(t\right)$, then the convolution theorem states that $Z\left(f\right) = X\left(f\right) \cdot Y\left(f\right)$. The convolution theorem goes both ways, so if $Z\left(f\right) = X\left(f\right) \star Y\left(f\right)$, then $z\left(t\right) = x\left(t\right) \cdot y\left(t\right)$.

The Discrete Fourier Transform (DFT) is what computational methods do, and its inverse, is defined as

$$X_k = \sum_{j=0}^{N-1} x_j \cdot e^{-2\pi i j k/N} \tag{9}$$

$$x_j = \frac{1}{N} \sum_{k=0}^{N-1} X_j \cdot e^{2\pi i j k/N} \tag{10}$$

where N is the total number of points in the signal. If the sample rate of the signal is f_s and the total sample time is $T = N/f_s$, then the time and frequency resolutions are $\Delta t = 1/f_s$ and $\Delta f = 1/T$, and the frequencies range from $-f_s/2$ to $f_s/2$ because of the Nyquist-Shannon sampling theorem. The method of efficiently and quickly computing the DFT is an algorithm called the FFT, which won't be discussed here, and will be taken for granted. The `numpy.fft` module contains functions that compute the FFT algorithm, as well as useful utilities.

One weird thing about the FFT is that the points in the spectrum for frequencies for 0 to $f_s/2$ come in the first half of the array that FFT returns and the spectrum for frequencies $-f_s/2$ to 0 is in the second half of the array. This can be more easily seen by looking at the output of the `numpy.fft.fftfreq` function, which returns the frequency values that correspond to the FFT return array. Similarly, the inverse FFT function returns the time samples from 0 to $T/2$ before the samples from $-T/2$ to 0. The function `numpy.fft.fftshift` can be used to arrange these arrays so that the zero points of the frequency or time axes correspond to the middle of the array, and it is a convenient tool.

That is a very quick summary of the Fourier Transform and the available methods of computing it in Python. It is not intended to be a complete explanation of, and mainly serves as an essential summary of the material that is important for computing the cross-correlation using the FFT.